第四章 树与二叉树

4.1树的基本概念

树的结点数等于所有结点的度数加1(总结点数=总分支数加1)

具有n个结点的m叉树的最小高度为脖 ⮦logm(n(m-1)+1)⮧

4.2二叉树的概念

空二叉树

空二叉树和度为2的有序树的区别

完全二叉树

1. i<<⮤n/2⮥,则结点i为分支结点，否则为叶子
2. 若n为奇数，则每个分支结点都有左右子女；

若n为偶数（编号为n/2），则编号最大的分支结点只有左子女，没有右子女，其余分支几点左右子女都有

二叉排序树

平衡二叉树

二叉树的性质：

1. N0=N2+1，N=N0+N1+N2

完全二叉树

1. i为奇数，其双亲结点为⮤i/2⮥,它是双亲结点的左孩子

i为偶数，其双亲结点为(i-1)/2,它是双亲结点的右孩子

1. 当2i<=N，结点i的左孩子编号为2i,否则无左孩子

当2i+1<=N,结点i的右孩子编号为2i+1,否则无右孩子

1. 结点i所在的层次（深度）为⮤log2i⮥+1
2. 具有n个结点的完全二叉树的高度为⮤log2n⮥+1或⮦log2(n+1)⮧

顺序存储结构？？

链式存储结构

typedef struct BiTNode{

ElemType data;

struct BiTNode \*lchild,\*rchild;

}BiTNode,\*BiTree;

//按顺序存储 在二叉树中查找结点i和j的最近公共结点??

ElemType Comm\_Ancestor(SqTree T,int i,int j){

if(T[i]!='#'&&T[j]!='#'){//结点存在

while(i!=j){//两个编号不同时循环

if(i>j)i=i/2;//向上找i的祖先

else j=j/2;//向上找j的祖先

}

return T[i];

}

}

4.3二叉树的遍历和线索二叉树

//因为每个结点都访问一次且只访问一次，故时间复杂度是O(n)

//递归工作栈的栈深恰好是树的深度

//先序遍历(PreOrder)

void PreOrder(BiTree T){

if(T!=NULL){

visit(T);

PreOrder(T->lchild);

PreOrder(T->rchild);

}

}

//中序遍历(InOrder)

void InOrder(BiTree T){

if(T!=NULL){

InOrder(T->lchild);

visit(T);

InOrder(T->rchild);

}

}

void InOrder2(BiTree T){

InitStack(S);BiTree p=T;//p是遍历指针

while(p||!isEmpty(S)){

if(p){//根指针进栈，遍历左子树

Push(S,p);//每遇到非空二叉树先向左走

p=p->lchild;

}

else{

Pop(S,p);visit(p);//根指针退栈，访问根结点，

p=p->lchild;//遍历右子树

}

}

}

//后序遍历(PostOrder)

void PostOrder(BiTree T){

if(T!=NULL){

PostOrder(T->lchild);

PostOrder(T->rchild);

visit(T);

}

}

//因为是先访问左子树，在访问右子树，最后访问根节点。所以当用堆栈来存储结点，

//必须分清返回根节点时，是从左子树返回的，还是从右子树返回的，

//所以借助辅助指针r，其指向最近访问过的结点。也可以在结点中增加一个标志域，记录是否已被访问

void PostOrder2(BiTree T){

InitStack(S);BiTree p=T;

BiTree r=NULL;

while(p||isEmpty(S)){

if(p){//走到最左边

Push(S,p);

p=p->lchild;

}

else{//向右

GetTop(S,p);

if(p->rchild&&p->lchild!=r){//如果左子树不空且未被访问过

p=p->rchild;//转向右

Push(S,p);

p=p->lchild;//再走到最左

}

else{//否则，弹出结点并访问

Pop(S,p);//讲结点弹出

visit(p);//访问该结点

r=p;//记录最近访问过的结点

p=NULL;//结点访问完后，重置p指针

}

}//else

}//while

}

/\*

//层次遍历

void LeVelOrder(BitTree T){

InitQueue(Q);

BiTree p;

EnQueue(Q,T);//根结点入队

while(!IsEmpty(Q)){

DeQueue(Q,p);//队头元素出队

visit(p);//访问当前p所指向结点

if(p->lchild!=NULL){//左子树不空，则左子树入队列

EnQueue(Q,p=>lchild);

}

if(p->rchile!=NULL){

EnQueue(Q,p->rchild);//右子树不空，则右子树入队列

}

}

}

\*/

线索二叉树：实质是对一个非线性结构进行线性化操作，每一个结点都有一个直接前驱和直接后继，所以引入一些空指针。

目的是：为了加快叉查找结点前驱和后继的速度

存储结构

typedef struct ThreadNode{

ElemType data;//数据元素

struct ThreaNode \*lchild,\*rchild;//左右孩子指针

int ltag,rtag;//左右孩子线索标志

} ThreadNode,\*ThreadTree;

0 lchid域指示结点的左孩子；1 lchild域指示结点的前驱

以这种结点结构构成的二叉链表作为二叉树的存储结构，叫做线索链表。其中指向结点前驱和后继的指针，叫做线索。加上线索的二叉树城为线索二叉树。对二叉树以某种次序遍历使其变为线索二叉树的过程叫做线索化。

线索二叉树的构造

对二叉树的线索化，实质上就是遍历一次二叉树，只是在遍历的过程中，

检查当前结点的左右指针域是否为空，若空，将它们改为指向前驱结点或后继结点的线索。指针pre指向中序遍历时上一个刚刚访问过得结点，用它来表示各结点访问的前后关系

void InThread(ThreadTree &p,ThreadTree &pre){

//中序遍历对二叉树线索化的递归算法

if(p!=NULL){

InThread(p->lchild,pre);//递归，线索化左子树

if(p->lchild==NULL){//左子树为空，建立前驱线索

p->lchild=pre;

p->ltag=1;

}

if(pre!=NULL&&pre->rchild==NULL){

pre->rchild=p;//建立前驱结点的后继线索

pre->rtag=1;

}

pre=p;//标记当前结点成为刚刚访问过得结点

InThread(p->rchild,pre);//递归，线索化右子树

}//if

}

//通过中序遍历建立中序线索二叉树

void CreateInThread(ThreadTree T){

ThreadTree pre=NULL;

if(T!=NULL){

InThread(T,pre);//线索化二叉树

pre->rchild=NULL;//处理遍历的最后一个结点

pre->rtag=1;

}

}

线索二叉树的遍历因为主要是为访问运算服务，所以不用借助栈，因为它的结点中隐含了线索二叉树的前驱和后继信息。

利用线索二叉树可以实现遍历的非递归算法,以下不含头结点的线索二叉树的遍历算法

//求中序线索二叉树中中序序列下的第一个结点 最后一个结点

ThreadNode \*FirstNode(ThreadNode \*p){

while(p->ltag=0)p=p->lchild;//最左下结点(不一定是叶结点)

return p;

}

ThreadNode \*LastNode(ThreadNode \*p){

while(p->rtag=0)p=p->lchild;

return p;

}

//求中序线索二叉数中结点p在中序序列下的后继结点 前驱结点

ThreadNode \*NextNode(ThreadNode \*p){

if(p->rtag=0)return FirstNode(p->rchild);

else return p->rchild;//1是后继 0是孩子

}

ThreadNode \*ForNode(ThreadNode \*p){

if(p->ltag=0)return ForNode(p->lchild);

else return p->lchild;

}

//不含头结点的中序线索二叉树中的中序遍历

void Inorder(ThreadNode \*T){

for(ThreadNode \*p=FirstNode(T);p!=NULL;p=NextNode(p))

visit(p);

}

//4.给出二叉树自下而上、从右到左的层次遍历算法

//5.求高度，采用二叉链表存储结构

int Btdepth(BiTree T){

if(T==NULL)return 0;

int ldep=Btdepth(T->lchild);

int rdep=Btdepth(T->rchild);

if(ldep>rdep) return ldep+1;

else return rdep+1;

}

//6.设二叉树中各结点的值互不相同，其先序和和中序遍历分别存于两个一维数组A[1...n]B[1...n]，试编写算法建立该二叉树的二叉链表

//利用先序找到根结点，在用中序递归左右子树，在判断长度

BiTree PreInCreat(ElemType A[],ElemType B[],int a1,int a2,int b1,int b2){

//a1 a2分别是第一和最后一个结点下标

BiTNode \*root;int i;

root=(BiTNode\*)malloc(sizeof(BiTNode));//建立根结点

root->data=A[a1];//根结点

for(i=b1;B[i]!=root->data;i++);//根结点在中序序列中的划分

int llen=i-b1;//左子树长度

int rlen=b2-i;//右子树长度

if(llen)//递归建立左子树

root->lchild=PreInCreat(A,B,a1+1,a1+llen,b1,b1+llen-1);

else

root->lchild=NULL;

if(rlen)//递归建立右子树

root->rchild=PreInCreat(A,B,a2-rlen+1,a2,b2-rlen+1,b2);

else

root->rchild=NULL;

return root;

}

//7.判断给定的二叉树是否是完全二叉树的算法。二叉树按二叉链表存储

bool IsComplete(BiTree T){

InitQueue(Q);

if(!T)return 1;//空树为满二叉树

while(!isEmpty(Q)){

DeQueue(Q,p);

if(p){//结点非空，将其左右子树入队列

EnQueue(Q,p->lchild);

EnQueue(Q.p->rchild);

}

else{//结点为空，检查其后是否有非空结点

while(!isEmpty(Q)){

DeQueue(Q,p);

if(p)return 0;

}

}

}

return 1;

}

//8.计算一颗给定二叉树的所有双分支结点的个数，采用二叉链表存储

f(b)=0 若b=NULL

f(b)=f(b->lchild)+f(b->rchild)+1 若\*b为双分支结点

f(b)=f(b->lchild)+f(b->rchild) 其他情况（\*b为单分支结点或叶子结点）

int DsonNodes(BiTree b){

if(b==NULL)return 0;

else if(b->lchild!=NULL&&b->rchild!=NULL)//双分支结点

return DsonNodes(b->lchild)+DsonNodes(b->rchild)+1;

else return DsonNodes(b->lchild)+DsonNodes(b->rchild);

}

//9.把树B中的所有结点的左右孩子进行交换，采用链式存储结构

void swap(BiTree b){

//递归地交换二叉树的左右子树

if(b){

swap(b->lchild);//递归交换左子树

swap(b->rchild);

temp=b->lchild;//交换左右孩子结点 BiTNode \*temp

b->lchild=b->rchild;

b->rchild=temp;

}

}

//10.先序遍历中第k(1<=k<=n)个结点的值，采用二叉链式存储结构

//11.删除二叉树中每个元素为x的结点，删去以它为跟的子树，并释放相应的空间，采用连输存储

//12.查找值为x的结点，并打印出他们的所有祖先(c)，（x<=1）

//13.

//14.

//15.

//16.

//17.

//18.

//19.

//20.

//21.

4.4树、森林

1. 树的存储结构
2. 双亲表示法

#define MAX\_TREE\_SIZE 100//树中最多结点树

typedef struct{//树的结点定义

ElemType data;//数据元素

int parent;//双亲位置域

}PTNode;

typedef struct{//树的类型定义

PTNode nodes[MAX\_TREE\_SIZE];//双亲表示

int n;//结点数

}PTree;

1. 孩子表示法
2. 孩子兄弟表示法

typedef struct CSNode {

ElemType data;

struct CSNode \*firstchild,\*nextsibling;//第一个孩子和有兄弟指针

}CSNode,\*CSTree;

1. 树、森林与二叉树的装换

左兄弟右孩子是相对二叉树而言

1. 树和森林的遍历

树 森林 二叉树

先根遍历 先序遍历 先序遍历

后根遍历 中序遍历 中序遍历

//5.求以孩子兄弟表示法存储的森林的叶子结点数

//6.以孩子兄弟链表为存储结构，递归算法求树的深度

//7.已知一颗树的层次序列以及每个结点的度，编写一个算法构造此树的孩子兄弟链表

4.5树和二叉树的应用

1. 二叉排序树
2. 定义
3. 查找
4. 插入
5. 构造
6. 删除
7. 查找效率
8. 平衡二叉树
9. 定义
10. 插入
11. LL(右单)
12. RR(左单)
13. LR(先左后右)
14. RL(先右后左)
15. 查找
16. 哈夫曼树和哈夫曼编码（Huffman）
17. 定义
18. 构造
19. 哈夫曼编码

//6.判定给定的二叉树是否为二叉排序树

//7.求出指定结点在给定二叉排序中的层次

//8.利用二叉树遍历思想编写一个判断二叉树是否为平衡二叉树

//9.求出给定二叉树中最小和最大的关键字

//10.从小到大输出二叉排序树中所有其值不小于k的关键字

//12.

//13.

//栈

typedef BiTree elemType;

#define MaxSize 50

typedef struct{

elemType data[MaxSize];

int top;

}SqStack;

SqStack S;

void InitStack(SqStack &S){

S.top=-1;

}

bool isEmpty(SqStack S){

if(S.top==-1)return true;

else return false;

}

//进栈判满 出栈判空

bool Push(SqStack &S,elemType x){

if(S.top==MaxSize-1)return false;

S.data[++S.top]=x;

return true;

}

bool Pop(SqStack &S,elemType &x){

if(S.top==-1)return false;

x=S.data[S.top--];

return true;

}

bool GetTop(SqStack S,elemType &x){

if(S.top==-1)return false;

x=S.data[S.top];

return true;

}